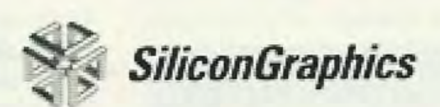
La primera revista de PC para los desarrolladores de videojuegos.



Por si lo dudábais, los dioses también fallan

John Carmak y sus amigos no son infalibles (¡qué alivio!); algunos listillos con ganas de hacerse notar han invertido su tiempo en el insano intento de "tumbar" a los servidores de Quake 2, y muy a nuestro pesar (y al de los numerosos usuarios de los servidores) lo han conseguido. ¿Cómo es posible? Pues ni más ni menos que enviando ciertos "paquetes bomba" IP. Éste es un trabajo para John Carmak y John Cash y sus superpoderes en los que a rutinas de comunicaciones y red se refiere.

Proyecto Fahrenheit



Microsoft



Una alianza importante: Microsoft y Silicon Graphics se unen para trabajar juntos en los que supondrá la revolución en el mundo de las tres dimensiones. Fahrenheit es el nombre clave, la idea es la de crear nuevos API´s 3D estándar con la idea de generar efectos tridimensionales a la altura de las estaciones gráficas Silicon utilizadas en las macroproducciones cinematográficas. Los resultados de la aplicación de Fahrenheit comenzarán a verse a principios del próximo milenio. ¿Cómo responderá el mercado ante semejante cambio?, tiempo al tiempo.

Sumario

Que no cunda el pánico, pero seamos prudentes

Todos conoceréis la noticia que hace unos días conmocionó al público durante la emisión en Japón de un episodio de dibujos animados manga, "Pokemon", tras el cual varios chavales japoneses sufrieron ataques de "epilepsia fotosensitiva", causados por la aparición en pantalla de repetidos flashes de luz roja y blanca. Pero ante todo, mucha calma, que no cunda el pánico. Para todos aquéllos que disfrutan alarmando a la sociedad decir que en todos los videojuegos, consolas, etc, se advierte claramente de las medidas que hay que tomar para que estos problemas no sucedan, así como las precauciones necesarias al hacer uso de ellos. Por lo que todo aquel que haga un buen uso de sus videojuegos disfrutará muchos años con ellos. PD: de todas formas se ha lanzado un aviso a todos los desarrolladores para que eviten, en la medida de lo posible, hacer uso de destellos o flashes que puedan resultar dañinos. Por cierto, Nintendo, creador del personaje Pichaku, asegura que su videojuego no tiene flashes ni blancos ni rojos. Nosotros les creemos.

Crea tus propios mapas para Quake 2 y Hexen 2

Gracias a la versión 1.6 del editor WorldCraft podréis crear vuestros niveles y mapas para Quake 2 y Hexen 2 sin necesidad de ninguna otra aplicación. Consigue esta nueva versión ya.

¡Quién da más!



Intel, el gigante de los microprocesadores, entra en combate. La guerra de los precios no ha hecho más que empezar. En plena campaña publicitaria de su flamante Pentium II, y ante la amenaza de sus competidores AMD y Cyrix (que reciben cada vez más apoyo de los fabricantes IBM y Compaq) han decidido tomar medidas y bajar los precios de sus micros Pentium y Pentium II entre un 20 y un 33 %. El sistema de competencia es bueno para el consumidor, mayor oferta, mayores posibilidades a la hora de elegir. Pero esta bajada de precios aún tardará un poco en llegar a nuestro país.

Inmersión TOTAL

Microsoft e Immersion Corp se destacan como pilares del hardware inmersivo, es decir, los nuevos joysticks con "retroceso" o "feedback". El Force FeedBack, de Microsoft, y el I-FORCE, de Inmersion Corp, ya son un estándar en la mayoría de los últimos videojuegos del mercado. Tras los joysticks y los ratones, ¿para cuándo cascos con feedback?

Destacamos

En nuestro CD de portada incluimos el siguiente material:

- **Las fuentes de código de los ejemplos comentados en 3D Manía.**
- **COOL EDIT: editor de samples para generar efectos de sonido de calidad.**
- **VISTA PRO: generador de mundos virtuales en 3D.**
- **Y la versión share para evaluación de Div Games Studio.**

Autor: Jesús de Santos García

Transformaciones geométricas

Las transformaciones geométricas juegan un papel fundamental en el movimiento de objetos y cámaras dentro de un mundo en tres dimensiones.

Vamos a dejar de momento el tema del pintado de polígonos (aunque profundizaremos en próximos artículos para profundizar en aspectos que todavía no hemos tocado, como por ejemplo interpolación RGB, Phong, Mapas de luces ...) para tratar de organizar un poco nuestro entorno 3d y dividirlo en objetos con movimiento propio e independiente de los demás.

Basta observar cualquier juego 3d para ver cómo cada entidad que se muestra en el mundo se mueve con respecto a un sistema de coordenadas independiente del mundo y de las demás entidades que aparecen en el juego. Así, el movimiento de avanzar hacia adelante un personaje está constantemente variando con respecto al mundo. Es lógico pensar que cada objeto tendrá un sistema de coordenadas propio. Necesitamos una herramienta que sea capaz de transformar de coordenadas locales del objeto a coordenadas de mundo.

Pero con las coordenadas de mundo todavía no hemos acabado. Necesitamos un paso más, pues el mundo se muestra en pantalla por medio de una cámara. Suele ser normal que la cámara de nuestro mundo se pueda mover (estamos hablando siempre de cámaras 6 DOF que permiten total libertad de movimientos). dentro del mundo como un objeto más. Por tanto la cámara poseerá su propio sistema de coordenadas. Será necesaria una transformación de coordenadas de mundo a coordenadas de la cámara. Finalmente estas coordenadas pueden ser ya proyectadas a pantalla de la forma que se explicó en el artículo pasado.

Vamos a dedicar el artículo de este mes a estudiar dos transformaciones geométricas básicas: la rotación en cada uno de los ejes y el desplazamiento en las tres posibles direcciones. Las matrices son una herramienta matemática que resuelven el problema planteado de una forma compacta y eficiente. Mucha gente teme a las matrices y emplean fórmulas intuitivas que en el fondo vienen de las matrices. Las matrices son un concepto superior que abarca la simple fórmula, y su perfecta comprensión nos permitirá emplear óptimamente algunas de sus propiedades más interesantes como la concatenación o la transformación inversa.

MATRICES

Un sistema de coordenadas en 3d debe constar de 3 ejes linealmente independientes de modo que cualquier coordenada del espacio que representa se pueda expresar como combinación lineal de estos ejes.

En nuestro caso vamos a añadir un vector más que indique el origen del sistema de coordenadas en el mundo de referencia.

Recordemos que el mundo de referencia es el mundo en el que se expresan las coordenadas de los ejes locales (en nuestro caso lo llamaremos mundo, aunque veremos el mes que viene que no siempre ocurre así). Las matrices que vamos a emplear son de la siguiente forma (ver figura 1).

Vamos a detallar poco a poco qué representa cada una de las variables. Lo primero de todo es comprender la aplicación que nosotros vamos a dar a la matriz. Podemos ver una matriz como una caja que toma datos de entrada y devuelve otros de salida. Esta 'caja tranformadora' lee puntos en 3d del sistema origen (local a partir de ahora) y nos devuelve puntos expresados en el sistema de referencia destino.

No existen, por tanto, coordenadas absolutas pues toda coordenada siempre estará expresada en función de una base de vectores. Nosotros de momento vamos a considerar el destino como el mundo cuyos vectores *x, y, z* son <1,0,0> <0,1,0> y <0,0,1> respectivamente, de tal modo que un vector dado *<x,y,z>* se expresa del mismo modo en este sistema de vectores.

El sistema de referencia local queda determinado por 3 vectores: uno para la coordenada x, otro para la y, y otro para la z. En la figura de la matriz:

- <x1,x2,x3> representa el eje de las X expresado en coordenadas de mundo.
- <y1,y2,y3> representa el eje de las Y expresado en coordenadas de mundo.
- <z1,z2,z3> representa el eje de las Z expresado en coordenadas de mundo.

Lo lógico es que el sistema de coordenadas local no esté centrado en el origen del mundo. Por tanto añadimos un vector más que indica la posición el coordenadas de mundo:

- <w1,w2,w3> representa el origen del sistema local expresado en coordenadas de mundo.

La cuarta columna de la matriz es necesaria para que nos quede una matriz cuadrada de 4x4.

Suponemos que el lector tiene conocimiento matemático de matrices y damos por sabido la multiplicación de matrices y conceptos como determinante de una matriz. Para transformar un vértice de coordenadas locales a coordenadas de mundo, multiplicamos la matriz vértice 1x4 por la matriz cambio de coordenadas de la figura 1.

Como la matriz cambio de base es 4x4 tenemos que emplear vértices en 4D para poder multiplicar por la matriz 4x4 (lo que se denomina coordenadas homogéneas). En la cuarta dimensión pondremos un 1.0 si el vector representa un vértice (y por tanto hay que tener en cuenta el desplazamiento) o pondremos un 0.0 si el vector representa un vector libre (no se tiene en cuenta su desplazamiento). Un ejemplo del primer caso son todos los vértices que representan una malla, mientras que ejemplos de segundo caso son los vectores que representan luz o los vectores normales a un

FIGURA 1. FORMA GENERAL DE LA MATRIZ CAMBIO DE BASE.

$$\begin{vmatrix} x1 & x2 & x3 & 0 \\ y1 & y2 & y3 & 0 \\ z1 & z2 & z3 & 0 \\ w1 & w2 & w3 & 1 \end{vmatrix}$$

FIGURA 2. TRANSFORMACION DE UN VERTICE POR UNA MATRIZ QUE DA UN VERTICE TRANSFORMADO.

$$\begin{vmatrix} vx & vy & vz & 1 \end{vmatrix} \begin{vmatrix} x1 & x2 & x3 & 0 \\ y1 & y2 & y3 & 0 \\ z1 & z2 & z3 & 0 \\ w1 & w2 & w3 & 1 \end{vmatrix}$$

plano. Emplearemos todo esto según vayamos construyendo nuestro engine.

Una propiedad muy importante y que todavía no hemos citado sobre la matriz cambio de base es que los vectores que representan los ejes X, Y, Z locales deben tener longitud 1 (es decir, que estén normalizados) y deben ser perpendiculares dos a dos. Matemáticamente se expresaría como que la submatriz 3x3 superior izquierda debe ser ortogonal. Hay dos razones básicas para emplear este tipo de matrices:

1. Las matrices de este tipo tranforman vectores conservando longitudes y ángulos. Esto quiere decir que por ejemplo la medida de un segmento local una vez tranformado a coordenadas de mundo se conserva. Es decir, no se producen deformaciones.
2. Se puede demostrar que la inversa de una matriz ortogonal es la transpuesta de la misma. (La transpuesta de una matriz se obtiene intercambiando filas por columnas)

Este último punto es el más importante. La inversa de una matriz cambio de base tranforma puntos de coordenadas de mundo a coordenadas locales (justo el proceso contrario que hemos estado comentando hasta ahora). El cálculo de la inversa de una matriz es costoso en tiempo y además no siempre tiene solución. Pero en el caso de una matriz ortogonal siempre existe solución y además ésta se obtiene simplemente transponiendo la matriz (cambio filas por columnas).

Esto que en principio puede parecer una tontería es básico para realizar simplificaciones importantes en el engine. Un ejemplo de esto (aunque trataremos todo esto mucho más a fondo) es la iluminación con un vector que represente la dirección de la luz. El vector luz se expresa en coordenadas de mundo. Para realizar operaciones con nuestros vértices locales tenemos que transformar a mundo todos los vértices y operar con la luz. Un método alternativo y mucho más eficiente es trasformar inversamente el vector luz a coordenadas locales y operar allí con cada vértice. La diferencia de rendimiento es clara.

Nuestro sistema de referencia local no es estático. Se puede mover por el mundo libremente y puede variar su orientación a lo largo del tiempo. Esto se consigue multplicando nuestra matriz por otras matrices que realicen las transformaciones adecuadas. La matriz resultado es nuestra nueva matriz. Este último paso es importante: nuestra matriz se va transformando de multiplicación en multiplicación mientras que los vértices siempre permanecen constantes.

Las operaciones básicas que vamos a tratar son rotación en los 3 ejes y desplazamiento:

Hay que resaltar el hecho de que todas las matrices de la figura 3 son ortogonales.

A la hora de multiplicar matrices podríamos construirnos una rutina general que multiplicase matrices pero esto no sería muy óptimo. Hay que aprovechar el hecho de que nuestra matriz tiene en la cuarta columna <0,0,0,1>. Para optimizar más aún debemos construirnos funciones que realicen las operaciones expresadas por las matrices de la figura 3.

Es importante resaltar que el producto de matrices no es conmutativo. En el uso que vamos a dar nosotros a las matrices podemos dar una interpretación clara:

- Una multiplicación por la izquierda opera sobre coordenadas del sistema de coordenadas origen. Si se quiere avanzar nuestro objeto (por ejemplo), debemos aplicar el desplazamiento en esta posición.
- Una multiplicación por la derecha opera sobre coordenadas ya transformadas a mundo.

Veamos esta diferencia con el caso más sencillo de transformación geométrica: el desplazamiento. Todas las fórmulas que aparecen en el listado 1, no son más que una simplificación del producto de matrices 4x4. La ventaja está en que una de las matrices (la de la izquierda en la primera función y la de la derecha en la segunda función) son conocidas y por tanto podemos simplificar las multiplicaciones por 1.0 y por 0.0. Es importante destacar que estamos trabajando con números en coma flotante y que las multiplicaciones son exactamente igual de costosas que las sumas. Por tanto no perdamos el tiempo en optimizaciones que eliminan multiplicaciones y que aumentan el número de sumas.

En el caso de la rotación, la simplificación es un poco más laboriosa. Pero tambien obtenemos un algoritmo mucho más eficiente que la simple multiplicación de matrices.

IMPLEMENTACION DE MATRICES

En el Cd-Rom que acompaña a la revista se incluye una clase de matriz con las funciones aquí comentadas más otras ampliaciones. Algunas cosas a destacar de esta implementación:

- Cada elemento de la matriz se expresa en coma flotante de 32 bits. (El tipo float de C++). Desde el Pentium, las operaciones en coma flotante son tan rápidas (o incluso más rápidas) como las operaciones con enteros. Es hora ya de olvidarse de la coma fija (*fixed point*) y trabajar con coma flotante que además ofrece una muy buena precisión. Tipos de mayor precisión serían excesivos para este problema.
- El cálculo de cosenos y senos se realiza en tiempo real y con las operaciones sin() y cos(). En princpio puede parecer tremendo pero todo depende del uso que se vaya a dar a estas funciones. Generalmente se realizan unas pocas tranformaciones geométricas por cada frame, lo cual representa un tiempo de proceso mínimo en comparación con el pintado total de la escena en cada momento. La FPU del Pentium tiene una instrucción que calcula el seno y el coseno simultáneamente (FSINCOS) que podría ser utilizada, pero en este caso no merece la pena el esfuerzo. Tampoco se va a ganar mucho rendimiento si pasamos a ensamblador estas rutinas. Es preferible 'perder' el tiempo en el *inner loop* del pintado de triángulo que en este apartado del engine.
- Ya hemos dicho antes que es importante que la matriz sea ortogonal. Tras muchas transformaciones debido a imprecisiones de la coma flotante nuestra matriz de cambio de base puede ir perdiendo poco a poco esa ortogonalidad. Esto se traduce en ligeras deformaciones de nuestros vértices e imprecisiones en transformadas inversas. Sería una buena idea corregir esta imprecisión (por ejemplo cada 100 frames) para evitar la pequeña molestia. Actualmente el engine del autor no realiza este chequeo y no ha encontrado ningún problema aunque también es cierto que no lo ha tenido funcionando mucho tiempo que es cuando se puede llegar a notar.

FIGURA 3. 1) ROTACION EN EL EJE X DE UN ANGULO T
2) ROTACION EN EL EJE Y DE UN ANGULO T
3) ROTACION EN EL EJE Z DE UN ANGULO T
4) DESPLAZAMIENTO (TX,TY,TZ).

1

1	0	0	0
0	cos t	sin t	0
0	-sin t	cos t	0
0	0	0	1

2

cos t	0	-sin t	0
0	1	0	0
sin t	0	cos t	0
0	0	0	1

3

cos t	sin t	0	0
-sin t	cos t	0	0
0	0	1	0
0	0	0	1

4

1	0	0	0
0	1	0	0
0	0	1	0
tx	ty	tz	1

El algoritmo de normalización de la matriz es sencillo:

1. Normalizamos los 3 vectores para que tengan longitud 1.
2. Tomamos 2 de ellos y los hacemos perpendiculares.
3. El tercer vector lo obtenemos del producto vectorial de los dos anteriores. (El producto vectorial de dos vectores es otro vector perpendicular a ambos).

CAMARAS

Hasta ahora nos hemos centrado en las transformaciones locales aplicadas a objetos. Las cámaras tambien se pueden expresar como una matriz ya que siguen siendo transformadoras de coordendas de un sistema origen (mundo) a un sistema destino (cámara a partir de ahora).

A la hora de rotar y desplazar la cámara hemos de multiplicar matrices por la derecha. Si lo hicieramos por la izquierda estaríamos



trabajando en coordenadas de mundo. Esta es una difrencia con las transformaciones anteriores donde multiplicábamos por la izquierda.
Otra diferencia importante es que el sentido de las rotaciones y de los desplazamientos es en sentido opuesto a la de los objetos. Si usamos las mismas funciones de tranformación que para objetos (que es lo lógico) deberemos cambiar siempre de signo. Esto queda claro observando el ejemplo que viene con el Cd-Rom.
Si el lector es capaz de resolver los siguientes dos puntos habrá comprendido a la perfección el tema del artículo de este mes:

1. Cálculo de la posición en coordenadas de mundo de un objeto a partir de su matriz. Lo que tenemos que hacer es transformar el punto <0,0,0> en coordenadas locales a coordenas de mundo. Pasamos el vector por la matriz y obtenemos la posición en coordenadas de mundo del origen del objeto. El problema se reduce simplemente a leer la cuarta fila de la matriz cambio de base.
2. Cálculo de la posición en coordenadas de mundo de la cámara a partir de su matriz. En este caso, no podemos resolver como en el punto 1. Para pasar de locales de la cámara a mundo (que es justo lo contrario de lo que hace la matriz cámara) hay que tranformar inversamente el vector <0,0,0> por la matriz cámara.

ORGANIZANDO NUESTRO ENGINE

Resumiendo, vamos a describir el camino que un vértice en coordenadas locales seguiría hasta llegar a 'convertise' en un píxel de pantalla:

1. El vértice en coordenadas locales nunca se modifica. Siempre tenemos su posición original que no varía. Todas las operaciones que se hagan sobre él se realizan sobre una copia. Recordemos que con este método los cambios se acumulan en la matriz y no en los vértices.
2. Se aplican las transformaciones locales (avance y desplazamiento). Esto se realiza multiplicando la matriz de transformación (por la izquierda) por la de cambio de base del objeto. Aquí si modificamos la matriz que va acumulando las tranformaciones. La matriz del objeto no conserva su valor inicial.
3. Transformamos el vértice a coordenadas globales.
4. Transformamos el vértice de coordenadas globales a cámara. Estos dos últimos pasos se pueden unir en uno multiplicando la matriz objeto por la de cámara y transformar por esta matriz. De todas formas, a veces puede ser necesario obtener la coordendas de mundo de un vértice.
5. Ya tenemos el vértice en coordenadas de cámara. Las tranformaciones que se realicen sobre la cámara se efectuan con multiplicaciones por la derecha. La matriz cámara tambien acumula valores y no se conserva su valor inicial.
6. Proyectamos el vértice a pantalla y ya tenemos un pixel 2d.

Hemos realizado un pequeño cambio sobre los ejemplos de esta sección. A partir de ahora vamos a trabajar con programas bajo Windows 95 y empleando DirectX con el fin de ponernos un poco al día. Si encontráis algun problema con esto o queréis que se dedique un artículo de esta sección a comentar algo sobre programación en Windows y DirectX orientada a juegos sólo tenéis que mandar un email al autor.
Con esto finalizamos el artículo de este mes. En el próximo seguiremos profundizando un poco más y añadiendo nuevos conceptos como la jerarquía entre objetos, la cinemática inversa.
Se agradecerían críticas, opiniones, sugerencias así como ideas sobre próximos temas a tratar en la dirección de Internet del autor. Hasta el próximo mes.

Jesús de Santos García
icg_ent@hotmail.com

LISTADO 1

```
// Desplazamiento multiplicando por la derecha. Se desplaza en coordenadas de mundo.

void matrix::translation(float dx, float dy, float dz) {

    mtx[3][0] += dx;
    mtx[3][1] += dy;
    mtx[3][2] += dz;

}

// Desplazamiento por la izquierda. Se desplaza en coordenadas locales.

void matrix::pre_translation(float dx, float dy, float dz) {

    mtx[3][0] += dx*mtx[0][0] + dy*mtx[1][0] + dz*mtx[2][0];
    mtx[3][1] += dx*mtx[0][1] + dy*mtx[1][1] + dz*mtx[2][1];
    mtx[3][2] += dx*mtx[0][2] + dy*mtx[1][2] + dz*mtx[2][2];

}
```

LISTADO 2

```
// Rotación en coordendas de mundo.
Multiplicación por la derecha.

void matrix::rotx(float angle) {

    float oldc;
    float cs=cos(angle),sn=sin(angle);

    oldc=mtx[0][1];
    mtx[0][1] = mtx[0][1]*cs - mtx[0][2]*sn;
    mtx[0][2] = mtx[0][2]*cs + oldc*sn;

    oldc=mtx[1][1];
    mtx[1][1] = mtx[1][1]*cs - mtx[1][2]*sn;
    mtx[1][2] = mtx[1][2]*cs + oldc*sn;

    oldc=mtx[2][1];
    mtx[2][1] = mtx[2][1]*cs - mtx[2][2]*sn;
    mtx[2][2] = mtx[2][2]*cs + oldc*sn;

    oldc=mtx[3][1];
    mtx[3][1] = mtx[3][1]*cs - mtx[3][2]*sn;
    mtx[3][2] = mtx[3][2]*cs + oldc*sn;

}

// Rotación en coordenadas locales.
Multiplicación por la izquierda.

void matrix::pre_rotx(float angle) {

    float oldc[3];
    float cs=cos(angle),sn=sin(angle);

    oldc[0]=mtx[1][0];
    oldc[1]=mtx[1][1];
    oldc[2]=mtx[1][2];

    mtx[1][0] = mtx[1][0]*cs + mtx[2][0]*sn;
    mtx[1][1] = mtx[1][1]*cs + mtx[2][1]*sn;
    mtx[1][2] = mtx[1][2]*cs + mtx[2][2]*sn;

    mtx[2][0] = -oldc[0]*sn + mtx[2][0]*cs;
    mtx[2][1] = -oldc[1]*sn + mtx[2][1]*cs;
    mtx[2][2] = -oldc[2]*sn + mtx[2][2]*cs;

}
```

Autor: Antonio Marchal

Scroll o movimiento de pantalla

En este artículo comentaremos los scrolls o movimientos de pantalla, así como sus ventajas e inconvenientes, aunque estos últimos sean pocos. Un buen scroll es lo que diferencia a un juego de otro en cuanto a calidad técnica se refiere, de ahí que éste sea un tema de gran importancia. Por eso, vamos a ver qué son, para qué sirven y cómo se usan.

Para conocer qué es un scroll vamos a poner un ejemplo. Imaginemos un mapa gráfico de 1000x1000; a la vez tenemos una ventana por donde veremos los gráficos de 100x100. Con estas medidas queda claro que, por la ventana, sólo veremos una décima parte del mapa gráfico. Al movimiento de esta ventana, posibilitando ver todo el gráfico, es a lo que se denomina scroll. Si sustituimos la ventana por la pantalla obtendremos, en vez de movimiento de ventana, movimiento de pantalla, que es otro de los nombres con los que se podría designar al scroll.

Un buen scroll es lo que diferencia a un juego de otro en cuanto a calidad técnica

Todo esto se traduce en que el observador sólo ve una parte del mapa en pantalla y para ver las otras partes se tiene que mover en la dirección contraria. Por ejemplo, si tenemos una calle con un muñeco dentro, e imaginamos que sólo se ve dos o tres casas de esa calle, cuando el muñeco avance hacia la derecha, la pantalla se moverá hacia la izquierda, quedando el muñeco centrado. Esto es lo que se denomina scroll de pantalla lateral o bidireccional.
Dentro de DIV Games Studio se pueden crear 10 de estos scrolls y cada uno de ellos puede tener dos planos, como quedará explicado más adelante.

¿PARA QUE SIRVEN?

El uso del scroll parece obvio. Cuando el mapa es mayor que la pantalla, el scroll sirve para poder ver así todo el mapa siendo éste el uso más común, si bien puede haber otros usos. Podemos hacer una pantalla cíclica utilizando scroll, es decir, que cuando se acabe el mapa por la derecha comience por la izquierda. En este caso no es necesario que el mapa gráfico sea mayor que la pantalla, ya que, cuando falte parte por cualquier lado completaremos la imagen con el otro. Este uso es un poco más "raro".
La última finalidad del scroll se puede comprobar en los fondos de un juego de coches. Tenemos una muestra en el ejemplo que incluye DIV Games Studio denominado "Speed for dummies". En el mismo, aparte de usar un modo 7 o modo abatido, para el horizonte se usa un scroll con una imagen de un paisaje. Esta mapa gráfico es cíclico, por este motivo, por mucho que gire el coche siempre se verá un fondo, completando las zonas que en principio tenían que estar vacías con la parte opuesta del gráfico (la del otro lado).
Aunque el mejor ejemplo de scroll dentro de DIV Games Studio, y en el que se puede ver su finalidad, podría ser *Helioball* o *Castle of Dr Malvado*. En el primero de ellos se pueden ver hasta dos scroll, uno por cada nave, mientras que en el caso del Dr. Malvado se puede comprobar el uso de los dos planos de scroll. Incluso en este ejemplo se hace uso de un tercer plano, que se crea usando una serie de técnicas que ya explicaremos.

¿COMO SE USAN?

Esta sección será, sin duda, la más extensa del artículo debido a que, aunque el uso de scroll en principio es fácil, existen una serie de matizaciones y modificaciones que complican el asunto. Se puede utilizar doble scroll, o también conocido como *split-screen*, se pueden crear scrolls con más de 2 planos de scroll, el scroll puede ser automático o manual, etc.
Para utilizar un scroll con DIV Games Studio, lo primero que hay que hacer es crearlo. Para ello se usa la función *start_scroll()*. A esta función, como a casi todas, hay que pasarle una serie de parámetros. El primero es el número de scroll, teniendo la posibilidad de crear hasta 10 scroll, con números del 0 al 9. Este número es bastante importante, ya que luego será al que hagamos referencia cada vez que se tenga que utilizar dicho scroll. Aparte, éste será el número que también usaremos al operar con la estructura scroll.
El siguiente parámetro es el código del fichero de gráficos donde están incluidos los usados como fondos para los planos de scroll. Tras esto aparece el número del gráfico usado como scroll principal, dentro del fichero nombrado anteriormente. Si se usara un segundo plano de scroll, el siguiente parámetro especificaría el número del gráfico que se usaría para dicho segundo plano.

Únicamente ve una parte del mapa en pantalla y para ver las otras partes la pantalla se mueve en la dirección contraria

Después viene el número de región, que hablaremos de ellas posteriormente ya que es necesario definirlas anteriormente. Y, por último, queda el indicador de bloqueo; éste señala si el scroll será rotatorio en alguna de las dimensiones, horizontal o vertical, de algunos de los dos planos posibles de scroll. Los valores de este último pueden ir de 0 a 15. Para más información sobre estos parámetros remitimos, como es habitual, a la ayuda del programa DIV Games Studio.
Aparte de los parámetros usados en la llamada a la función *start_scroll()* existe una tabla que integra 10 estructuras, una para cada uno de los posibles scrolls. Cada una de estas estructura incluye varios campos y variables a las que podremos tener acceso en la forma habitual a la que accede DIV Games Studio a las estructuras; por ejemplo para manipular la variable *z* del scroll número 3 y cambiar su valor a 234, se debería incluir la siguiente línea:

```
scroll[3].z=234;
```





Funciones y variables comentadas en el artículo

start_scroll(<nº scroll>,<fichero>,<grafico primer plano>,<grafico segundo plano>,<region>,<indicador_bloqueo>)

Aunque está suficientemente comentada dentro del artículo, aquí se dará una breve descripción de todos sus parametros. <Nº de scroll>, es el número de scroll a definir, de los 10 disponibles. <Fichero> es el fichero donde se encuentran los gráficos de los planos del scroll. Hay que tener en cuenta que los dos gráficos deben estar en el mismo fichero. <Gráfico primer plano> y <gráfico segundo plano>, son los números de cada uno de los gráficos de cada plano dentro del fichero seleccionado. <Región> es el número de la región donde aparecerá el scroll; si es distinto de cero, habrá que definir primero esa región. <Indicador de bloqueo>, señala si el scroll es cíclico en alguna de las coordenadas de alguno de los planos.

Ctype y cnumber

También comentadas en el articulo, éstas son variables locales de las que dispone cada uno de los procesos. La primera indica que las coordenadas del proceso que posea dicha variable se encontrará dentro de un scroll o modo 7, mientras que la segunda indica dentro de cuáles de los scrolls se debe incluir.

Pero el orden de uso es indistinto en los campos que la integran, como es obvio. Empecemos, por ejemplo, por "*z*". Esta variable determina el plano de impresión del scroll, es decir, si se imprimirá por debajo o por encima de los procesos cuya "*z*" sea distinta. En la variable "*camera*" se debe introducir el código identificador del proceso al que sigue la cámara, en caso de que se quiera un scroll automático. De las posibilidades de scroll automático o manual hablaremos más adelante. Todos los parámetros usados en estas estructuras, a excepción *de "z"*, *"x0","y0","x1" e "y1"*, sólo son útiles cuando se haga uso del scroll automático. Por el contrario, las cuatro últimas variables nombradas únicamente tienen uso en los scrolls manuales. Por otro lado, "*ratio*" maneja la velocidad relativa del segundo plano respecto al primero en tanto por ciento. Es decir, si se usa un valor de 200 significará que el segundo plano irá la mitad de rápido que el primero. Como es obvio, el uso de esta variable tendrá sentido cuando se usen dos planos de scroll, aparte de la necesidad de que sea automático, como se ha dicho anteriormente. "*Speed*" manejará la velocidad máxima a la que se moverá el primer plano de scroll, por defecto está a 0, lo que no impone ningún limite. Pero se podría especificar el número máximo de puntos que se permitan mover el scroll en modo automático. Las dos siguientes variables, "*region1*" y "*region2*" están íntimamente unidas. La primera indica la región a partir de la cual el scroll tendrá efecto. Por defecto su valor es –1, lo que implica que no hay ninguna región definida, y si se definiera una región, el proceso que hace las veces de cámara, cada vez que saliera de dicha región provocaría un movimiento de la misma. La segunda región limita el espacio del cual el proceso cámara no saldrá por ninguna razón. Normalmente está inicializado a 0, que es la región de la pantalla completa. Por último, quedan "*x0*", "*y0*", "*x1*" e "*y1*" que controlan las coordenadas horizontal y vertical del primer y segundo plano, respectivamente. Su utilidad, como anteriormente se ha señalado, únicamente se da en el caso de que los scrolls sean manuales.

Aparte de esta tabla con las estructuras de los scrolls y de la función *start_scroll()*, existen un par de variables locales de las que dispone cada proceso para definirse como dentro de un scroll. Estas son "*c_type*" y "*c_number*"; la primera sirve para designar al proceso del cual se defina el valor de esta variable como dentro del scroll. Es decir, que sus coordenadas serán relativas al scroll en vez de a la pantalla. Para incluir un proceso le debemos pasar el valor constante "*c_scroll*", de la manera:

ctype=cnumber;

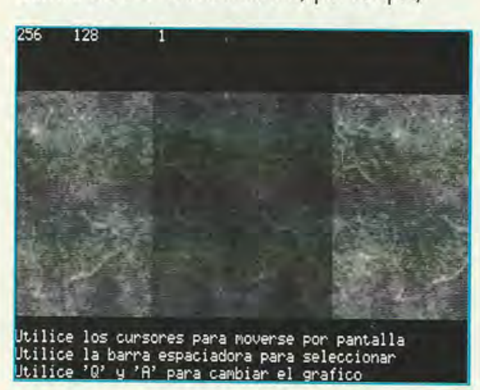
La otra variable "*c_number*" sirve para indicar dentro de cuáles scrolls de los creados debe aparecer el proceso. Su valor por defecto es 0, que indica que el proceso se incluirá en todos los scrolls. Para señalar que sólo aparezca en algunos existen unas constantes que se formar con la letra "*c*", seguida del subrayado y del número de scroll donde queremos incluir al proceso. Por ejemplo, para incluir un proceso únicamente en los scrolls 3 y 5,deberíamos inicializar su variable "*cnumber*" de la siguiente forma:

c_number=c_3+c_5;

Estos son todos los datos que hacen referencia al scroll. Muy someramente, para que un scroll tenga efecto deberíamos seguir los siguientes pasos: primeramente crear el scroll con la función *start_scroll()*, y, después, si se quiere que el scroll sea automático, crear un proceso que será al que siga la cámara. Para que se cumpla esto, tendremos que pasar el código identificador del proceso a la variable camera, de la estructura de scroll correspondiente. Al proceso que hace de cámara y a todos los incluidos en el scroll, deberemos ponerles el valor correspondiente en "*ctype*" y "*c_number*", aunque en la última variable solamente cuando fuera necesario.

Se ha nombrado a lo largo del artículo el concepto de región. En DIV Games Studio existe la posibilidad de crear regiones de pantalla, para ello se debe usar la función *define_region()*. Pasando como parámetros, primeramente, el número de región a crear, seguido de las coordenadas iniciales y el tamaño de la ventana. Si se incluye un proceso dentro de una región, el gráfico de dicho proceso únicamente aparecerá en dicha región. Por defecto, todos los procesos están incluidos en la región 0 que es la propia pantalla al completo. Pero, la pregunta sería: ¿qué tiene que ver el scroll con las regiones? En primer lugar, para los scrolls automáticos se pueden definir regiones para cuadrar el movimiento de la cámara, como antes se ha mencionado. En segundo lugar, si se quieren crear varios scroll, normalmente se define una región para cada scroll creado. De hecho, a la hora de crear el scroll se debe pasar como parámetro el número de región donde aparecerá dicho scroll. Normalmente cuando se tiene un solo scroll, se suele pasar la región 0 que es la pantalla completa.

Después de ver todos estos fundamentos en cuanto scrolls, pasemos a la práctica. Veamos casos especiales como, por ejemplo, el doble scroll o *split screen*. Para crear este tipo de scroll de doble pantalla se tienen crear dos regiones, una para cada scroll, ya que al llamar a la función *start_scroll()*, para crear cada uno de los scrolls, se debe pasar como parámetro el número de región que hayamos elegido para él. Después, se deben crear los procesos que hagan las función de cámara si el scroll va a ser automático. Un ejemplo práctico de este tipo de uso de scrolls se hace en el juego *Helioball*, que viene de ejemplo dentro de DIV Games Studio, por lo que,



Estructura scroll[]

Aunque está comentada dentro del artículo es posible encontrar más información en la completa ayuda de DIV Games Studio, por ello, únicamente se dará una pequeña descripción de cada campo para posibles consultas:

Z = Profundidad de impresión del scroll.
Camera = Identificador de la cámara del scroll.
Ratio = Velocidad relativa entre planos de scroll.
Speed = Velocidad máxima de segundo plano.
Region1 = Región donde se empieza a hacer el scroll.
Region2 = Región donde no se deja que pase el scroll.
X0,y0 = Coordenadas del primer plano de scroll.
X1,y1 = Coordenadas del segundo plano de scroll.

Define_region [nº de region, x , y , ancho , alto]

Define una región de pantalla. Como parámetros se deben pasar <nº de región>, que es el número que luego usaremos para hacer mención a esa región. <X> e <Y> que son las coordenadas de inicio de la región. Y, por ultimo, <ancho> y <alto>, que es el tamaño de dicha región.

desde aquí, recomendamos su estudio para una mejor compresión.

Para utilizar un scroll con DIV Games Studio, lo primero que hay que hacer es crearlo; para ello se usa la función *start_scroll()*

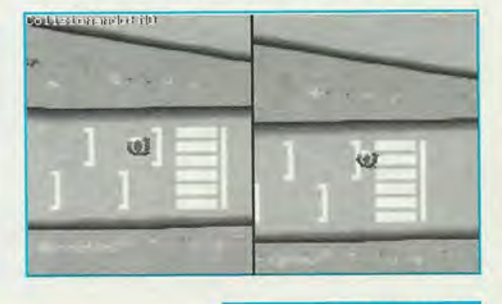
Otro de los temas importantes es crear más de los dos planos de scrolls permitidos. Un uso de esta función se hace dentro del juego *Castle of Dr. Malvado,* que también aparece en DIV Games Studio como ejemplo. En él se usan los dos planos posibles, uno para el fondo y otro para el decorado en sí. Pero, esta vez, aparece un tercer plano por encima de los demás, con distintos objetos de decorado, como tumbas, plantas, etc... Para conseguir este efecto se ha creado el scroll de manera normal, siendo automático. Luego se crea un proceso, que es el que imprime el tercer plano, cuya función es coger las coordenadas actuales del scroll y posicionar los gráficos en relación a estas coordenadas. Para que esto tenga un efecto más realista y no aparezcan los objetos con cierto retraso, lo que se hace es llamar a la función *refresh_scroll()* que permite conocer las coordenadas de scroll actualizadas del scroll, lo que posibilita una impresión perfecta, no teniendo coordenadas "antiguas". Otras cuestiones a tener en cuenta a la hora de crear un tercer plano, es el plano de profundidad donde se imprimirá dicho plano. Este valor lo podemos modificar con la variable local "*z*", que posee el proceso que va a hacer las funciones de tercer plano de scroll. Con este método se podrían incluir un cuarto plano, quinto, sexto, etc. Otro ejemplo de esta capacidad de tercer plano se encuentra, de manera más simplificada que dentro del juego de *Castle of Dr Malvado*, en la ayuda del propio DIV. Para acceder al mismo debemos mirar la página de ayuda, destinada a *refresh_scroll()*, y ejecutar el programa que incluye dicha página. Recomendamos esta lectura antes que la del juego *Castle of Dr. Malvado,* ya que en la página de ayuda es mucho más sencillo y compresible que el otro. Sólo hay que recordar que el mes pasado hablamos de mapas de durezas y uno de los ejemplos que lo usaban era el mismo *Castle of Dr. Malvado*, por lo que el listado de dicho programa resulta bastante interesante, como se puede comprobar.
Uno de los problemas que podemos encontrar es cuando se haga uso de la posibilidades de la variable local "*cnumber*" y se tengan que detectar colisiones. ¿Se detectarán las colisiones de dos gráficos que colisionan si no están dentro del mismo scroll? La respuesta es no y tiene su lógica, ya que si en uno de los scrolls no vemos el gráfico del otro proceso, no parece muy útil detectar un choque con dicho proceso. De todas formas, en caso de urgente necesidad se podría detectar con otras funciones de DIV, como *get_dist(), fget_dist(), get_distx()* o *get_disty(),* pues todas estas funciones devuelven la distancia entre dos procesos o puntos. Lo único que tenemos que hacer para saber si a habido una colisión es coger la distancia que devuelven esta funciones y comprobar si es menor que la máxima, si los dos procesos estuvieran tocándose. Un ejemplo parecido a esto lo podemos encontrar en el juego *Speed for dummies*, otro de los que aparecen en DIV Games Studio, pero con la diferencia de que, aunque el juego es en modo 7, donde no de pueden detectar colisiones, debe utilizar estos métodos para poder detectarlas entre los coches.

Otros de los temas importantes es crear más de los dos planos de scrolls permitidos

Otra de las posibilidades al hacer un scroll es realizarlo usando lo que se conoce en el mundo de los programadores como *tiles.*, que traducido sería "movimiento de pantalla con losetas". En este caso, en vez de crear el gráfico del scroll completo, se crean una serie de losetas o *tiles* para elaborar dicho mapa. Luego se debe crear lo que se conoce como un mapeador que consiste en un programa que nos permitirá colocar dichas losetas dentro del mapa. Pero vamos a explicar el funcionamiento de esta forma de hacer scroll mas explícitamente. Para conseguir que el programa "recuerde" qué losetas poner, debemos codificarlas, es decir, asignar un número a cada tipo de losetas. Luego, en el mapeador, iremos colocando estas losetas cuyo valor asignado se guardará en una tabla que usaremos en el programa que utilice ese mapa. Por ejemplo, creamos 20 losetas y le asignamos números del 1 al 20, con una medida de, por ejemplo, 20 x 20 píxeles. Y vamos a crear un mapa de 10 x 10 losetas, por lo tanto, el gráfico de scroll final será de 200x200 píxeles, que sale de multiplicar el número de losetas por su ancho o alto, dependiendo de la coordenada utilizada. Por otro lado, se tiene que crear una tabla con 100 elementos (resultado que se obtiene de multiplicar 10 losetas por 10 losetas) para guardar los valores de cada losetas. Después, para acceder a una loseta dentro de la tabla, es decir, convertir las coordenadas "*x*" e "*y*" a una posición, usaremos la formula:

map_block_copy(<fichero>,<gráfico destino>,<x_destino>, <y_destino>,<gráfico origen>,<x_origen>, <y_origen>,<ancho>,<alto>)

Es usada por los mapa tileados para copiar gráficos. Los parámetros necesarios son: <fichero> que es código del fichero donde se encuentran los gráficos a copiar. Hay que indicar que, al igual que pasa con la función start_scroll(), tanto el gráfico destino como el de origen, deben encontrarse dentro del mismo fichero. <Gráfico destino> es el número de gráfico dentro del fichero anterior. <X_destino> e <y_destino> son las coordenadas donde se copiara el gráfico. <Gráfico origen> es el número de gráfico dentro del fichero utilizado. <X_origen> e <y_origen> son las coordenadas desde donde se cogerá el gráfico. Finalmente, <ancho> y <alto> son el tamaño del gráfico a copiar.



refresh_scroll(<nº de scroll>)
Esta función es utilizada para hacer un refresco del scroll. El único parámetro necesario es el número de scroll a refrescar. Esta función es necesaria porque los gráficos que se copien con cualquier de las funciones destinadas para ellos, no aparecerán hasta que se mueve el scroll. Para entender mejor esto, en el ejemplo del mapeador de tiles se hace uso de esta función. Desde aquí invitamos a quitarla para ver qué ocurre....

load(<nombre de archivo>,<offset dato>)
Esta función carga del disco un archivo con datos a una variable. Los parámetros utilizados son <nombre de archivo>, que es el nombre con el que lo hemos grabado, y <offset dato>, que es la posición dentro del programa donde se encuentra la variable donde queremos guardar los datos. Esta posición se puede hallar poniendo delante de la variable la palabra reservada offset o bien el símbolo &, que es lo mismo que offset a todos los efectos.

save(<nombre de archivo>,<offset dato>,<longitud dato>)
Esta función esta ligada a load() y sirve para guardar datos de una variable dentro del disco duro. Los parámetros necesarios son el nombre del archivo, la posición de memoria dentro de programa de la variable a grabar, que se puede conseguir utilizando la palabra offset o el símbolo &, y, por último, la longitud del dato, que se puede calcular usando la palabra reservada sizeof(), a la cual, si se le pasa el nombre de una variable, devuelve su tamaño. Hay que tener precaución al comprobar dónde se graba o carga un archivo.

*Numero_en_la_tabla= x_mapa+ (y_mapa*ancho_mapa);*

En el caso anterior, por ejemplo, para acceder a la loseta que está situada en la posición del mapa cuya "*x*" es 4, y cuya "*y*" es 5 usando la formula anterior, daría 4+(5*10)=54, que es la posición en la tabla donde guardaremos el valor de la loseta en cuestión. ¿Fácil, no? Un aspecto del que no se ha hablado todavía es cómo usar en el juego la tabla que hemos creado y, por lo tanto, el scroll de *tiles*. Para llevar a cabo este cometido nos encontramos con las funciones *save()* y *load()*, que graban y cargan datos respectivamente. Entonces, para usar el scroll se construye con el mapeador y se guardan los valores en la tabla para, una vez terminado el "trabajo", grabarlos en disco usando la función ya nombrada.
Posteriormente, en el juego se carga con la función *load()* la tabla que habíamos guardado y en un gráfico creado aparte, con el tamaño del mapa de scroll, en el ejemplo anterior, 200x 200, se pegan los gráficos de los *tiles* usando la función *map_block_copy()*. Esta función se usa normalmente también en el mapeador para ir visionando los *tiles* que vamos poniendo y así ir creando el scroll.
De esta manera, se pueden construir mapas de scroll inmensos ya que únicamente se debe guardar la tabla y los gráficos de los *tiles*.
Además, si cambiamos la tabla o los tiles, el scroll se modificará automáticamente. Para guardar los *tiles* se pueden usar dos métodos: guardar los *tiles* por separado, es decir una por gráfico, o juntarlos todos en un gráfico y usar la función *map_block_copy()*, únicamente de la zona que nos interese, es decir, en la posición del *tile* elegido, copiar el tamaño del mismo. Veamos esto último con un ejemplo. Si se quieren tener 32 *tiles*, de 40 por 40 puntos, se podría, si se usase el primer método, grabar cada uno separadamente, mientras que usando el segundo método de guardar *tiles* se crearía una pantalla de 320 por 160. Y se pondrían los *tiles* dentro de la misma, en 8 columnas por 4 filas (si se comprueban los cálculos, veremos que concuerdan, 320=8x40 y 160=4x40). Luego, como la función *map_block_copy()* necesita necesariamente las coordenadas de origen desde donde se copia el gráfico, sólo resta designar estas coordenadas dependiendo del *tile* elegido.

LOS EJEMPLOS

Como es habitual, dentro del CD que se incluye con la revista aparecen varios ejemplos para clarificar los conceptos del artículo; en esta ocasión son 3 los ejemplos que se incluyen. El primero de ellos nos mostrará una pantalla partida por la mitad, con lo que podremos ver las posibilidades de *split screen* o doble scroll. Pero este ejemplo incluye otra serie de cuestiones, pues tanto el scroll de la derecha como el de la izquierda tienen otro par de diferencias. En primer lugar, uno de ellos se hace de manera manual y el otro es cíclico. ¿Sabéis a simple vista cuál es el automático? Bueno, el automático es el que centra automáticamente el coche, y no se dispersa por los lados como le pasa al scroll manual. La otra diferencia entre los scroll es que uno de ellos es cíclico en sus dos coordenadas, es decir, cuando falta pantalla por un lado, lo coge del contrario para completar.
El segundo ejemplo es un mapeador de *tiles* muy sencillo. En pantalla aparecen las teclas usadas para moverse y situar los *tiles*, así como las coordenadas donde estamos posicionados en ese momento, además del código de la loseta seleccionada. Al iniciarse el programa, carga el mapa que tuviéramos grabado en ese momento, y al salir lo graba automáticamente. Si por cualquier causa DIV Games Studio diera un error al comienzo de la ejecución de este programa, seguramente será debido a que no encuentra el fichero con los datos del mapa. Este fichero debe estar, en principio, en el directorio donde está el ejecutable de DIV Games Studio, el D.EXE. Esta ubicación es cambiable, y os insto a hacerlo como prueba, para así evitar tener demasiados ficheros que no tienen nada que ver uno con otro y están dentro de un mismo directorio. Lo que no se especifica aquí es el uso de un solo mapa gráfico para todos los *tiles*, pero esto también se dejará para futuras modificaciones que podéis hacer (así aprendéis un poco mas de programación, que nunca viene mal).
Por último, el tercer ejemplo hace uso de las posibilidades de "*c_number*" y sirve para comprobar cómo no se detectan las colisiones entre dos procesos que se encuentran en distintos scroll. El aspecto es similar al primer ejemplo. Esta vez, los dos ejemplos son de scroll automático y cíclico. Aparte, cada uno de los coches es un proceso distinto, con igual gráfico pero con distinto nombre. Como en el caso anterior, invito a los lectores a modificar las variables locales "*c_number*" de cada uno de los procesos, para ver qué pasa y comprobar, así, cuándo se detectan colisiones y cuándo no.

HASTA LUEGO, LUCAS...

Bueno, con esto se acaba el artículo de este mes, esperando que no se nos haya quedado nada en el tintero en cuanto a scrolls. Si alguien tiene alguna dudad, ya sabéis dónde podéis mandar vuestras cartas y e-mails, aunque deseamos que todos los aspectos de los scrolls os sean de utilidad. Y que os DIVirtais programando. Hasta el mes que viene...

get_dist(<código identificador>)
Esta función sirve para hallar distancias; para utilizarla se le pasa el código identificador del proceso destino, llamando a la función desde el proceso origen.

fget_dist(<x0>,<y0>,<x1>,<y1>)
Es muy parecida a get_dist() pero, en este caso, se le deben pasar las coordenadas de dos puntos, de los cuales se hallará la distancia.
get_distx(<ángulo>,<distancia>) y get_disty(<ángulo>,<distancia>)
Estas dos funciones son similares, de ahí que vengan juntas. Como parámetros se le deben pasar un ángulo y una distancia, y la función devolverá la distancia en la coordenada x y en la coordenada y, respectivamente.





(*) DESPUÉS DEL 31 DE ENERO DE 1998

De lo contrario, puedes perder una oportunidad de ORO: conseguir completamente GRATIS una versión REGISTRADA de DIV GAMES STUDIO.

EL PRIMER ENTORNO PROFESIONAL PARA LA PROGRAMACIÓN DE VIDEOJUEGOS • INCLUYE UN PAQUETE DE 16 JUEGOS COMPLETOS • GRAN CANTIDAD DE GRÁFICOS Y SONIDOS • HERRAMIENTAS DE DISEÑO • UN COMPLETÍSIMO MANUAL • Y MUCHO MÁS...

Y todo ello por el mismo precio de la suscripción tradicional. Vamos, a que esperas, corre a cumplimentar el cupón y entra YA en el mundo de la programación profesional.

(*) OFERTA VÁLIDA HASTA AGOTAR EXISTENCIAS

Autor: Jorge Rosado

Introducción al Desarrollo

Este artículo está dedicado a todos aquellos que deseáis emprender el largo camino que supone la creación de un videojuego, pero aunque os sobra valentía y empeño puede que os falte la experiencia y sabiduría necesarias para comenzar con muy buen pie. Para todos ellos ofrecemos ideas sobre la creación del grupo de desarrollo ideal.

No hace falta ser una lumbrera para, a poco que os paréis a pensar, descubráis que para la creación de un videojuego (o de cualquier cosa en este mundo) hacen falta 2 cosas esenciales, sin las que sería absurdo el plantearse tal creación. La primera de ellas es el autor, y la segunda, evidentemente, la obra.

Pues bien, en el caso que nos atañe, el desarrollo de un videojuego, el papel del autor (en este caso de los autores) recae en el denominado grupo de desarrollo (otro término muy utilizado y completamente erróneo es el de grupo de programación. Bienaventurado el que piensa que un videjuego es sólo cosa de programadores, pues comete uno de los mayores errores de su vida), mientras que el papel de la obra lo representa, sin duda, el propio videojuego.

Por tanto ya tenemos el primer enigma resuelto: para comenzar la tarea necesitamos conseguir un grupo de desarrollo y diseñar un videojuego.

EL GRUPO

Empecemos haciendo un poco de historia sobre los grupos de desarrollo. Desde los tiempos del Spectrum y demás ordenadores de 8 bits, los videojuegos han dominado los corazones de millones de adolescentes. Fruto de esta pasión fueron las toneladas de títulos que abarrotaban las listas de lanzamientos por aquel entonces. La mayoría de ellos realizados de forma casi artesanal (vamos casi POKE a POKE).

Por aquel entonces los grupos de desarrollo eran muy reducidos. Era muy posible que una sola persona hiciera sus propios juegos, que se ocupara tanto de la programación como de los gráficos, e incluso es posible que de la mismísima música. El resultado de todo esto, excepto en contadas ocasiones, eran juegos que solían flaquear en algún aspecto (fantásticos gráficos, pero malísima programación, o viceversa), pero el factor común de todos ellos era su gran originalidad. A pesar de todo, en aquella época ya despuntaban los genios que con el paso del tiempo se convertirían en los puntales del mundo del desarrollo de videojuegos de hoy en día.

Para comenzar la tarea necesitamos conseguir un grupo de desarrollo y diseñar un videojuego

Pero lo verdaderamente importante del asunto es que el presupuesto necesario para realizar un videojuego comercial era bastante bajo (si lo comparamos con los actuales) y debido al reducido número de personas que participaban en la realización se podían hacer casi desde casa. En este momento se acuñó el termino de FreeLance, que representa a los que trabajan por libre sin necesidad de acudir al trabajo.

LA PROFESIONALIZACION DEL GRUPO

Pero los tiempos en los que un videojuego se podía realizar en casa junto con unos cuantos amigos se terminaron hace mucho. Los grupos, digamos familiares, poco o nada tienen que hacer contra las auténticas factorías de videojuegos (tanto en el terreno de la producción como en el de la distribución), en las que el nivel de profesionalización alcanza cotas elevadísimas, dando como resultado videojuegos de mayor calidad.

Al igual que las superproducciones cinematográficas, en el extranjero se cuentan con presupuestos cuantiosos para realizar títulos rompedores que recauden el doble o más del dinero invertido. Por tanto, en la época que

Profesionalización

Aquí tenéis los 4 pasos esenciales que debe tomar todo grupo de desarrollo para comenzar su periodo de profesionalización.

1. Centralizar el desarrollo

Un videojuego consta de numerosas partes (código, gráficos, ...) todas ellas interdependientes entre sí. Para facilitar o incluso posibilitar el proceso de creación se debe centralizar el trabajo en un solo punto. Todos los grupos de desarrollo profesionales cuentan con un centro de trabajo en el que se reúnen sus componente a la hora de trabajar en el videojuego.

2. Fijar unos objetivos

El grupo siempre debe tener el rumbo fijo. La obra a realizar debe permanecer clara en la cabeza de todos los integrantes del grupo. Para ello, se deben fijar unos objetivos en cuanto a magnitud y características de la obra, en este caso el videojuego en cuestión, para que durante el transcurso del desarrollo se cambie de rumbo lo menos posible, pues evidentemente en una tarea como ésta las modificaciones siempre se presentan.

3. Fijar un calendario de trabajo

Un videojuego no puede convertirse en una Torre de Babel ni en la catedral de la Almudena, ya que no se convertirá en auténtico videojuego hasta que esté totalmente terminado. Los cementerios de proyectos inacabados están llenos de cadáveres de videojuegos que se comenzaron sin tener en cuenta la fecha de su finalización y se alargaron y alargaron hasta morir o quedar desfasados.

4. Aceptar responsabilidades

Aunque esté orientado al mercado del ocio y el divertimento, el desarrollo de un videojuego profesional no se debe considerar como un pasatiempo o una afición. Los integrantes del grupo deben aceptar la responsabilidad que da el participar en un proyecto común en el que todos los miembros del grupo trabajan para conseguir crear una obra maestra. Sin embargo, esto no debe significar que se pierda la ilusión en el trabajo, sino que se desee que el trabajo esté bien hecho.

vivimos, el proyecto de creación de una desarrolladora de videojuegos debe realizarse paso a paso si se pretende comenzar desde cero, para, poco a poco, ir acercándose cada vez más a las potencias inglesas o americanas.

EL GRUPO IDEAL, ¿UTOPIA O REALIDAD?

Sencillamente, y sin temor a equivocarnos, afirmaremos que no existe una estructura de grupo ideal. Pero precisemos, la confección de un grupo de desarrollo depende mucho del tipo de videojuego que se quiera abordar, tanto en número de integrantes como en las características de cada uno de sus componentes.
Respecto a las características necesarias de los miembros del equipo, pongamos como ejemplo el desarrollo de una aventura gráfica actual al estilo *Broken Sword II* (casi una película de dibujos animados) o *Riven* (render 3D puro y duro), y el desarrollo de un arcade 3D al estilo *Quake*. Si dividimos las "tribus" dentro de un grupo de desarrollo de la siguiente forma: programadores, grafistas, diseñadores y músicos podremos afirmar que a la hora de realizar el *Broken Sword II* necesitaremos un grupo con una alta carga de grafistas (bocetistas, animadores, fondistas, ...) mientras que las tareas de programación no necesitan de los conocimientos avanzados de matemáticas, física, ..., que son indispensables para realizar un engine 3D para un juego tipo *Quake*.

Una aventura como Broken Sword no necesita de programadores que sepan lo último en física como ocurre en Quake

En lo que respecta al número de integrantes del grupo ideal debe decidirse atendiendo al tiempo que se requiere para realizar el proyecto.
Éste es un terreno bastante escabroso a la hora de decantarse por alguna teoría válida referente a la relación numérica entre los integrantes del grupo.
Un elevado número de personas en un grupo de desarrollo puede acelerar los tiempos siempre y cuando se consiga una gran coordinación entre todos ellos. Pero no nos dejemos engañar por las apariencias, en incontables ocasiones (y sobre todo con integrantes noveles) el refrán "cuanto menos bulto, más claridad" resulta profético.
Con un grupo bien asentado y que trabaje bien, un número entre 7 y 8 personas puede resultar ideal para abordar la mayoría de géneros del mercado. Repartidos de la siguiente manera: 1 programador principal (engine, física, lógica, ...) 1 programador de apoyo (red, sonido, IA) 1 programador de herramientas (editores, conversores, compresores, instaladores) 1 diseñador (guión, niveles, mapeado) 1 grafista principal (diseño artístico, modelado, animaciones, textureado) 2 grafistas de apoyo (modelado, textureado, animaciones) 1 músico (banda sonora, efectos de sonido)

ULTIMOS CONSEJOS

En resumen, podemos concluir que para entrar en el mundo del desarrollo de videojuegos con buen pie y con amplias posibilidades de alcanzar los puestos de los más grandes, hace falta plantearse la cuestión como un proyecto profesional. El negocio de los videojuegos cobra cada día más importancia en el mundo de la informática, y tanto fabricantes como distribuidores apoyan cada vez más con su producto a este mercado. Un buen grupo de desarrollo, con moral y coordinacion, que tenga las cosas claras y los objetivos fijados tanto a corto como a largo plazo, puede llegar a emular a los monstruos del desarrollo actual.

Grupos históricos

Éstos son algunos de los grupos de desarrollo que han marcado una época en la historia de los videojuegos.

Era 8 bits: Ultimate ¡Play the Game!

Uno de los primeros casos de grupo familiar. Los geniales hermanos Stamper nos deleitaron en la época de los 8 bits con joyas como Jet Pack, Knight Lore, Underworld... Y, a pesar de desaparecer del mapa durante un tiempo, son uno de los pocos ejemplo de perfecta adaptación a las nuevas tecnologías. Ahora responden al nombre de Rareware, os suenan (Donkey Kong Country, Killer Instinct, ...)

Era Amiga: Bitmap Brothers

Los Bitmap Brothers fueron los reyes del arcade en la época del Amiga. Autores de títulos clásicos como los inmejorables matamarcianos Xenon y Xenon 2, o los primeros deportivos futuristas SpeedBall (también 1 y 2) les lanzaron a la fama y tras éxitos para PC como el Chaos Engine les permitieron realizar el estratégico Z (opiniones de todo tipo).

Era PC: Id Software

Simplemente los amos del Universo. Tras asomarse al Universo de los videojuegos con la serie Commander Keen, los hermanos Carmak y John Romero asombraron al mundo con bombazos del calibre de DOOM y Quake. Ahora son multimillonarios y conducen deportivos. Un ejemplo más sobre la teoría del paso a paso (ahora bien en el mundo a poca gente como John Carmak, espero).

Configuraciones según proyectos

Estas son algunas de las configuraciones sugeridas como grupo de desarrollo ideal, poniendo como ejemplo alguno de los generos más interesantes del momento.

Arcades 3D

Debido a la complejidad de la programacion será necesario un número de programadores elevado. Es indispensable que al menos uno de ellos esté bien cualificado y posea experiencia en la programación 3D. Sin embargo, gracias a las actuales aceleradoras 3D, el potencial gráfico de estos juegos cada vez es más exigente por lo que no hay que descuidar el diseño de los personajes y sus movimientos.
2-3 Programadores: Principal, apoyo, herramientas
2-3 Grafistas: Diseño, modelado, textureado
1-2 Diseñadores: Niveles y Mapeado
1 Músico
Total: 6- 9 personas

Estratégicos

1-2 Programadores: Principal (potenciando IA y Red), apoyo (editores flexibles)
3-4 Grafistas: Modelado personajes, escenarios, menús, marcadores
3-4 Diseñadores: Niveles, Mapas, Estrategias
1 Músico
Total: 8 - 11 personas

Aventuras

1 Programador
3-8 Grafistas: Diseño Personajes y escenarios, Fondos, Animadores, Retoque
1 Diseñador: Guión y diálogos
2 Músicos: Banda sonora, efectos, voces
Total: 7 - 12 personas

Autor: Jorge Rosado

La figura del Diseñador

Desde esta plataforma informativa queremos gritar a los cuatro vientos la siguiente información (y citamos para ello a John Romero, gurú entre gurús): " El DISEÑO es ley". Dentro de poco sabrás porque nos exaltamos de esta manera.

Si señor, la figura del diseñador de videojuegos, empañada durante años por la de los programadores estrella, parece recobrar cada día más y más fuerza en el esquema del mundo del videojuego. Cada vez se presta más atención y se tiene más en cuenta al diseñador que se oculta tras la obra, al cerebro que imprime su personalidad en el juego, en definitiva, al motor que desplaza la totalidad de la maquinaria de desarrollo hacia el éxito (o por qué no, hacia el fracaso).

JUEGOS CON PERSONALIDAD PROPIA

Sin ninguna duda, existen juegos que destacan por encima de los demás, que van más allá de las alaracas técnicas, las rutinas 3D, y demás pamplimas, que no necesitan de envolturas preciosistas con pantallas de render, gráficos Silicon, ..., porque basan todo su poderío en conceptos originales, argumentos elaborados y una jugabilidad fuera de límites.

Pero no penséis que, por ello, reniegan de las innovaciones tecnológicas y gráficas, ni mucho menos; simplemente lo que sucede es que en estos juegos toda la parafernalia técnica se utiliza como herramienta para recrear entornos, personajes y situaciones de la manera más realista posible. Éste es uno de los grandes secretos que encierran los éxitos de ventas: consiguen potenciar el JUEGO, sin ocultarlo entre sus características técnicas.



DAVE PERRY.

Al igual que el Dr. Frankestein, ellos consiguen dar vida a lo que nadie puede

Nadie disfruta con un engine 3D por muy tecnológico que sea, pues solamente son polígonos con texturas que se desplazan más o menos rápido. Pero si estos engines se orientan bien y se comienzan a construir niveles inteligentes, personajes carismáticos y se consigue una gran interactividad entre ellos, tendremos un gran juego.

Aquí es donde un buen diseñador define. Es el único capaz de dar vida a los inertes polígonos y conseguir que cientos de jugadores se apasionen con su juego. Al igual que el Dr. Frankestein, ellos consiguen dar vida a lo que nadie puede.

Alexei Pajitnov

Compañía: Spectrum Holobyte
Cargo: Diseñador
Localización: Alameda, California, USA

Desde la mismísima Rusia (en concreto desde el Computer Center of the Moscow Academy of Science) nos llega el creador del Tetris. Su desarrollo es sencillo: debes apilar formas geométricas más o menos complejas para ir eliminando líneas. Juego responsable de miles de casos de obsesión geométrica. Nintendo, no podían ser otros, lo compraron. Posteriormente, se instaló en Estados Unidos para trabajar en las filas de Spectrum Holobyte.

Dave Perry

Compañía: Shiny Entertainment
Cargo: Presidente y Programador principal
Localización: Laguna Beach, California, USA

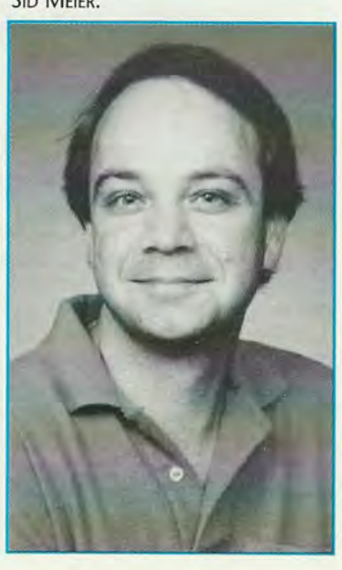
Dave Perry ha amasado una fortuna con su compañía Shiny Entertainment, gracias a juegos como EartwormJim o Cool Spot, originarios consolas 16bits, pero convertidos a la mayoría de formatos de la tierra. Comenzó en las filas de Virgin y fundo su compañía propia compañía. Shiny, que se ha convertido en grupo oficial de desarrollo de Interplay, ha cosechado éxitos para PC de la talla de MDK. Ahora preparan títulos para las nuevas consolas de 32bits, Wild - 9, RC Copter, ..., serán los juegos a tener en cuenta en los próximos meses. Dave Perry listo, atractivo y rico. ¡Quién da más!

Sid Meier

Compañía: MicroProse (ahora Firaxis)
Cargo: Fundador y Diseñador
Localización: Hunt Valley, USA

Siendo el cofundador de MicroProse, Sin Meier no ha estado interesado en escalar puestos en su compañía para mantenerse en el frente del desarrollo de los juegos en los que ha participado. Solamente mencionar un nombre, Civilization, y las puertas del cielo se abren a su paso. El juego de construcción de imperios por excelencia. Le han seguido Colonization, Civilization II, y el nuevo Gettisburg ! (su sueño personal) para su nueva compañía Firaxis.

SID MEIER.




Chris Roberts.

Chris Roberts

Compañía: Origin
Cargo: Productor Ejecutivo
Localización: Austin, Texas, USA

Después de las sagas de la Guerra de las Galaxias y Star Trek, la siguiente frontera se sitúa en el universo de Wing Commander. Chris Roberts dio vida en 1989 al primer Wing Commader de la historia hasta la fecha (con un nutrido número de secuelas que se acercan cada vez más a las películas antes mencionadas) la serie Wing Commander ha vendido más de 2 millones de copias en todo el mundo. Una cosa: este chico cada vez está más interesado en el mundo del cine; que se prepare James Cameron.

LOS GURÚS DEL DISEÑO

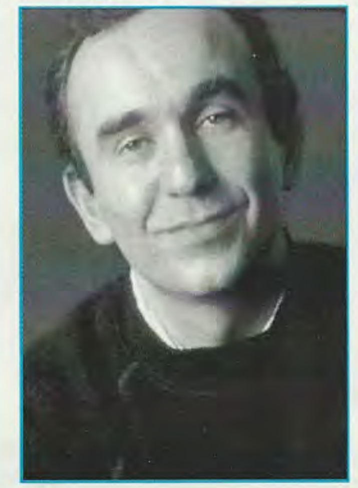
Los hermanos Stamper, Dave Perry, Peter Molineux, ..., son los nombres (conocidos por todos) de los responsables de algunos de los mejores y más jugables videojuegos de la historia. Si te interesa el tema y quieres conocer un poco más de los adnegados personajes que se esfuerzan por salir de la corriente de juegos mediocres que arrastra y arrastrará mercado, no pierdas de vista a los 8 gurús que te presentamos, pues siguen en activo y aún les quedan muchos ases en la manga.

Una imaginación poderosa, y millones de años de experiencia acumulada en el mundo de los videojuegos, son los responsables de hacer fáciles las cosas que a los demás nos parecen difíciles, de apreciar sutiles detalles que aumentan la jugabilidad de un producto hasta limites insospechados. Todos ellos son ejemplos a seguir, pero si tuviéramos que destacar a uno entre los demás nos decantaríamos por Peter Molineaux. Desde Bullfrog lleva años haciendo muescas en su revolver y almacenando premios y mas premios. Los denominados juegos de simulación de Dios, no existirían de no haber sido por la aparición del gran Populous, o la serie Theme.
Pero no es oro todo lo que reluce es posible que a pesar del apogeo del diseño, los diseñadores se conviertan en una especie en extinción. La causa de este fenómeno no es otra que la propia industria dominada por auténticos gigantes que pueden arrollar a su paso todo ápice de creatividad y rebeldía amparándose en cifras de ventas y balances contables. Esperemos que esto no suceda nunca y digamos ...

EL DISEÑO ES LEY

Si la racha sigue y los desarrolladores no se dejan impresionar por las aceleradoras 3D, el MMX, los Pentium II,..., para pasar a fijarse más en el diseño de los juegos, todos saldremos ganando. Aunque no lo parezca, las nuevas tecnologías abren nuevos campos para explorar géneros de videojuegos, mejorar las actuales, y crear, crear, crear, ... Todos a una : ¡El DISEÑO es ley!

Peter Molyneux.



Al Lowe

Compañía: Sierra On-Line
Cargo: Diseñador
Localización: Chicago, USA

Si hablamos de personajes carismáticos, no podemos dejar de mencionar a Larry Laffer, el gigolo en paro protagonista de la serie Leisure Suit Larry, una de las que más beneficios ha reportado a la compañía de las aventuras gráficas por antonomasia Sierra. Se incorporó a la empresa en 1982 y comenzó a trabajar en el King Quest 3 con Roberta Williams, realizando su primer proyecto en solitario en 1987. Leisure Suit Larry in the Land of the Lounge Lizards había nacido.

Roberta Williams

Compañía: Sierra On-Line
Cargo: Cofundador y Diseñadora
Localización: Chicago, USA

Roberta y su marido Ken fundaron On-Line Systems, (después añadirían el nombre de Sierra y pasarían del Systems) en 1980.
Su primer juego Mystery House. A partir de entonces los éxitos se sucedieron y, junto con Lucas, Sierra es la reina de las aventuras gráficas. King Quest, Larry, Quest for Glory, Phantasmagoria, ... Un consejo no perdáis de vista el último King Quest (el 8), una obra maestra.


John Romero.

John Romero

Compañía: Id Software (ahora ION Storm)
Cargo: Cofundador y Diseñador
Localización: Mesquite, Texas, USA

¿Cuál ha sido el juego que ha permanecido en más ordenadores durante más tiempo? Unánimemente el DOOM. El primer arcade 3D de calidad, al que siguieron múltiples clones, la mayoría de ellos bastante casposos. El único problema de este personaje es que su ego supera a su ingenio y le trae algún que otro problema. Tras el desarrollo de Quake para Id, John ha salido de esta compañía para crear ION Storm, siguiente proyecto Daikatana.

Peter Molyneux

Compañía: Bullfrog (antes LionHead)
Cargo: Fundador y Director
Localización: Guildford, Inglaterra.

El dios de los juegos de estrategia (modalidad Dios): la serie Populous (ya por su tercera entrega), Theme Park, Theme Hospital, la serie Magic Carpet, Syndicate y Syndicate Wars, y su último y más reciente éxito Dungeon Keeper. Con un aspecto similar al del Amo del Calabozo y una expresión impasible, Peter Molineaux observa al resto de los mortales subido en su trono desde el que dirige a los integrantes de sus equipos. Creó un nuevo género de juego, el simulador de Dios con Populous; no hay duda Peter se define como IMAGINACION. Desde aquí le deseamos suerte en su nueva etapa con LionHead.

Autor: Víctor Segura

Banda sonora Paso a Paso

En este número de la revista vamos a comentar y explicar los diferentes pasos necesarios para la elaboración de la banda sonora de un videojuego. Pasos aplicables tanto a composiciones realizadas en entornos MIDI como composiciones realizadas en TRACKERS.

Aunque la realización de una banda sonora no es una tarea puramente mecánica, conviene formalizar de alguna manera los diferentes pasos a seguir desde que comenzamos el desarrollo musical del videojuego hasta que hacemos la grabación final a CD de las canciones que hemos compuesto.
A continuación daremos los diferentes pasos con una breve descripción de lo que hay que hacer en cada uno de ellos.

ESTILO MUSICAL

Lo primero que debemos hacer es encontrar el estilo musical que mejor se adapta al videojuego. Para encontrarlo hemos de fijarnos tanto en el aspecto gráfico del juego como en el lugar y tiempo en el que la acción de este se desarrolla.
Como ya dijimos anteriormente en esta revista, la relación estilomusical - juego es la siguiente: Si el ambiente es futurista, entonces podemos enfocar nuestra música hacia un estilo Techno, Orquestado, Newage o Rock. Supongamos que el juego efectivamente tiene una ambientación futurista y es de carreras, (tipo WipeOut o Thunder-Offshore). En este caso lo importante es dotar de cierto dinamismo y "velocidad" al aspecto sonoro del juego. El Techno se adecuaría perfectamente a nuestras necesidades. Supongamos ahora que la ambientación sigue siendo futurista, pero esta vez es un juego del estilo de la Guerra de las galaxias. Nuestra intención es adentrar al jugador en un ambiente más tranquilo y envolvente, razón más que suficiente para utilizar un estilo más "peliculero" con ayuda de nuestra propia orquesta (con un buen módulo de sonidos, tendremos a nuestra disposición la mejor de las orquestas), aunque una ambientación newage (al más puro estilo Vangelis) puede ser más que suficiente para que el jugador disfrute y se introduzca con mayor facilidad en el ambiente galáctico del videojuego. Finalmente supongamos que el juego tiene como objetivo principal acabar y destruir todo lo que aparece en pantalla (tipo Duke-Nukem y Quake). Unas guitarras con distorsiones "terroríficas" acompañadas de un buen bajo y una batería "cañera" bastarían para conseguir subir el nivel de adrenalina del jugador.

Para un juego de acción no hay nada como unas buenas guitarras distorsionadas

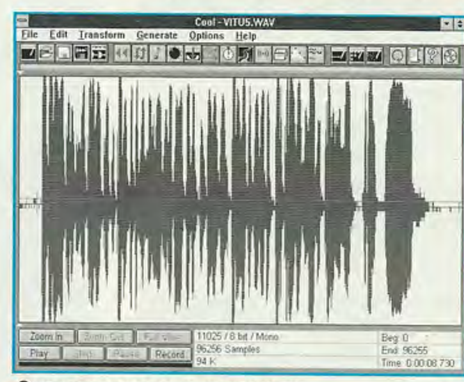
Del mismo modo, sea cual sea el tiempo en el que se desarrolla la acción del videojuego, tenemos la siguiente relación:
Para carreras de coches (tipo fórmula 1 o Indy): Rock.
Para juegos épicos (con castillos, dragones, ...): Música Orquestada.
Para aventuras gráficas: Ambientaciones sonoras (acordes tipo "camas" acompañados de sutiles melodías).
Para juegos tipo tetris: Melodías pegadizas.
Para juegos de lucha: Diferentes ambientaciones dependiendo del lugar donde se combate. El estilo ha de ser trepidante (Rock por ejemplo), utilizando instrumentos típicos de los países en los que se desarrolla el combate.



ONDA ORIGINAL.

¿UNA SOLA CANCION?

Es importante pensar cuantas canciones vamos a componer para dosificar así nuestro esfuerzo en un tema principal (tema insignia del juego) y las diferentes melodías relacionadas con las diversas situaciones que se presentan en el juego o en componer, por el contrario, tantas canciones como fases tenga el juego. Esta decisión depende principalmente de la estructura del propio juego. Imaginemos que el juego es de carreras de coches. Como hemos dicho anteriormente, el estilo que utilizaríamos sería Rock. Al ejecutar el juego podemos encontrarnos con una intro, a continuación el menú principal (y los diferentes menús de elegir coche, circuito, etc...) y luego sabemos que el juego consta de 8 circuitos diferentes. En este caso, es lógico pensar en una canción para la intro, otra para los menús, y otra para cada uno de los circuitos. En total tendríamos que desarrollar 10 canciones.
Imaginemos que el juego tiene cierto ambiente épico, y queremos que un tema principal sea la base o la insignia del aspecto musical del juego. Podemos hacer diferentes canciones para las fases dentro del estilo que hayamos elegido, y en el momento en el que aparece el enemigo final de cada fase que suene el tema principal, que es el que aparece también en la intro y en los menús del juego. Así, tendríamos que preocuparnos más en la elaboración de dicho tema principal ya que su aparición a lo largo de la partida sería prácticamente continua. Esta técnica se utiliza mucho en películas (Indiana Jones por ejemplo), de manera que siempre que el héroe aparece en un momento importante suena el tema principal de la banda sonora. Así pues, la decisión de realizar temas diferentes o un solo tema principal con uno o



ONDA CON EFECTO DE DISTORSION.

dos temas más de menor importancia, depende directamente de la estructura del juego y de las ganas y del tiempo del que disponemos para realizar la banda sonora.

MELODIA, ACOMPAÑAMIENTO E INSTRUMENTOS

Cuando hemos elegido el estilo musical y el número de canciones que vamos a componer, ya estamos preparados para realizar la banda sonora del juego.
A la hora de componer una canción cada uno tiene su propio método. Se puede pensar en una melodía y luego hacer el acompañamiento y el resto de los instrumentos y de los efectos sonoro-ambientales del la canción. O, por el contrario, empezar por una base rítmica y en función de dicha base ir desarrollando el resto de la canción.
Es muy frecuente que la melodía principal que habíamos pensado aunque este bien hecha, no suene exactamente como habíamos pensado. Un manera interesante para solucionar este problema es la de incluir algún efecto tipo flanger, reverb, eco o cualquier otro efecto al instrumento que lleva el peso de la melodía principal. Si aún así no hemos solucionado el problema, o incluso suena peor que al principio, lo que podemos hacer es que otro instrumento diferente haga lo mismo, esto es, "desdoblamos" la melodía y en vez de usar un instrumento principal para dicha melodía utilizamos dos o más diferentes. Las plantillas instrumentales (los instrumentos que vamos a utilizar en una canción) dependen en general del estilo de la canción. Si queréis saber más al respecto, en el número 3 de GAME OVER dedicamos un apartado a este tema en concreto.

AJUSTES FINALES

Una vez terminada nuestra canción nos enfrentamos a la tarea más ardua y trabajosa: los ajustes finales.
Para empezar debemos escuchar los diferentes canales de la canción con sus variadas instrumentos uno a uno, para ajustar así los volúmenes y el brillo de dichos instrumentos individualmente. Por ejemplo supongamos que en una canción Rock tenemos bajo, batería y guitarra. Lo que tenemos que hacer es desactivar todos los volúmenes menos el de un canal (por ejemplo el del bajo). A continuación escuchamos el bajo (a lo largo de toda la canción) y hacemos las pequeñas o grandes modificaciones necesarias, como subir o bajar el volumen de determinadas notas, ajustar el brillo global del instrumento, la velocidad de ejecución de las notas, etc.... Una vez hecho esto con cada instrumento hacemos lo mismo aumentando el número de instrumentos con el volumen activado, esto es, activamos el volumen del bajo y como hemos visto anteriormente suena como queríamos; a continuación activamos el volumen de la batería y ajustamos los volúmenes globales de los dos instrumentos. Finalmente activamos el volumen de la guitarra y hacemos lo mismo en relación con los volúmenes del bajo y de la batería. Y así sucesivamente hasta acabar con todos los instrumentos de la canción. Puede que este proceso tengamos que repetirlo alguna que otra vez en determinadas partes de las canciones en las que queremos resaltar un determinado instrumento respecto del resto de los instrumentos, pero no debemos desesperarnos, ya que todo este "trabajo interno" de la canción nos ayudará a obtener un resultado satisfactorio. Este mismo proceso ha de hacerse con el balance de cada instrumento, así como con los efectos (reverb, chorus, flanger, etc...) que hayamos utilizado en la canción. Una vez terminado este último proceso lo único que tenemos que hacer es grabar nuestra canción con calidad digital, esto es, guardar el archivo (con extensión midi, mod, s3m o cualquier otra) en un archivo WAV, a 16 bits y 44100 Hz.

COMENTARIO FINAL

Es importante disponer de una base musical, pero también es importante (y mucho) oír los diferentes trabajos de otras personas tanto de bandas sonoras de videojuegos como de películas, estudiar dichas bandas sonoras, criticar el trabajo que uno hace como si fuera de otro músico, analizar hasta el último detalle los instrumentos de cada canción (por qué se ha utilizado un determinado instrumento en un momento en concreto, qué instrumentos y con qué efectos se utilizan para determinadas ambientaciones, etc..).
Porque como dice Hartmut Zeller, tan importante es escuchar como estudiar.

RECOMENDACIONES

He aquí una lista de bandas sonoras que aparecen en diferentes videojuegos que os recomendamos que escuches e incluso que analicéis.

Estilo Techno:	Wipe Out
	Thunder offshore
Estilo New Age:	Magic Carpet
Estilo Ambiental:	Blade Runner
	Warcraft2
Estilo Rock:	Need for speed I y II
	Estilo "Rock-Consolero"
	Battle Arena Thoshinden
	Estilo Rock-Alternativo
	Quake
Estilo Orquestado:	Tie fighter

Por supuesto que hay muchos videojuegos con muy buena música y tal vez mejor que los aquí mencionados; esto es solo una referencia de los diferentes estilos que podemos encontrar en un videojuego.

REFERENCIAS

Muchas veces nuestras capacidades creativas se ven limitadas por nuestros conocimientos, que en determinados casos pueden llegar a ser bastante limitados. Aquí tenéis una serie de libros que pueden ayudaros a ampliar dichos conocimientos desde el punto de vista teórico.
Orquestación. de Walter Piston. Este libro trae ejemplos y ejercicios resueltos de cómo armonizar diferentes melodías, para orquestas enteras, cuartetos de cuerdas, metales, etc..
(Imprescindibles conocimientos musicales de lectura de partituras).
Teoría Musical y Armonía Moderna Vol.I y II. de Enric Herrera. Los dos volúmenes tratan de teoría musical, escalas, funciones tonales, armonía, etc...
Técnicas de Arreglos para la orquesta moderna. de Enric Herrera. Para aprendices de John Williams.
Para los que deseen saber algo más acerca de los diferentes instrumentos y sus diferentes estilos tenemos:
Blues Hanon, Jazz Hanon, Rock Hanon. Tres libros para los pianistas en los que podrán encontrar una breve reseña teórica de cada estilo y una gran cantidad de ejercicios prácticos.
Hard Rock Combination Studies. Por Michel Fath. Con técnicas y ejercicios para guitarristas de hard Rock.

Autor: Víctor Segura

Los efectos y sus parámetros

¿Para qué sirven los parámetros de los diferentes efectos que se pueden aplicar sobre sonidos, voces o instrumentos? ¿Para qué demonios sirve y qué es el feedback? Este mes analizaremos los parámetros principales de efectos como el reverb, el delay, el chorus, el compresor y el flanger.

Disponer de un buen procesador de efectos (ya sea en software o en hardware) está al alcance de todos. Pero ¿realmente aprovechamos al máximo cada uno de los efectos que incorpora nuestro procesador? Probablemente no, por eso vamos a describir los diferentes parámetros de los efectos más usuales, que podemos encontrar tanto en procesadores de efectos como en módulos de sonidos.

COMPRESOR

El primero de los efectos a estudiar es el compresor. Es un efecto limitador que suprime los pocos transitorios de ataque de alto nivel de señal de entrada, por compresión de dicha señal. Con este efecto aumentar el *sustain* del sonido o mejorar el énfasis en el ataque de dicho sonido. Los parámetros principales son:

- *Sens* (sensibilidad): Mide el grado del efecto compresor. A mayor sensibilidad, más se enfatiza el ataque o el *sustain*.
- *Attack* (ataque): Velocidad del ataque. A mayor valor, más rápido es el ataque.
- *Level* (nivel): Nivel de salida del sonido procesado.

DISTORSION/OVERDRIVE

Estos dos efectos crean un efecto de distorsión típica. La distorsión es un efecto de saturación de alto nivel de ganancia, y el *overdrive* es una saturación más suave. Los parámetros principales son:

- *Drive* (saturación): Cantidad de distorsión o ganancia.
- *Tone* (tono): Tono del sonido procesado. A mayor valor, más brillo en el sonido del efecto.
- *Level* (nivel): Nivel de salida del sonido procesado.

Si lo que se pretende es obtener una distorsión tipo metálico, aumentar ligeramente el tone y subir el valor del drive.

CHORUS/FLANGER

Estos dos efectos sirven para crear un sonido cálido y ondulante, por modulación continua del tono de la señal de entrada. Los parámetros principales son:

- *Speed* (velocidad): Velocidad de la modulación
- *Depth* (profundidad): Profundidad de la modulación. A mayor profundidad más profundidad en el efecto.
- *Feedback* (realimentación): Cantidad de realimentación. Los valores altos crean una modulación más profunda para el flanger. Los valores bajos crean un efecto chorus más limpio.
- *Mix* (mezcla): Ajusta el nivel de la mezcla entre sonido puro y sonido procesado.

DELAY

El *delay* simula el efecto de eco que repite sonidos. Los parámetros principales son:

- *Time* (tiempo de retardo): Tiempo de retardo (medido en msg) entre el sonido directo y el sonido procesado.
- *Feedback:* Cantidad de realimentación. A mayor realimentación, más número de veces se produce la repetición o eco del sonido.
- *Mix* (mezcla): Ajusta el nivel de mezcla entre sonido puro y procesado.

REVERB

En general hay 5 tipos de reverb: Hall, ensemble hall, sala, metal y escenario.

Este efecto tiene muchos parámetros. El parámetro principal es *Mix* (mezcla), que ajusta el nivel de mezcla entre sonido puro y procesado.



Onda con efecto de delay.

El *reverb* es, por lo general, el efecto más utilizado, tanto en la música sintética, como en actuaciones. Proporciona más cuerpo al sonido (simulando por ejemplo que el sonido está sonando en una habitación vacía "hall"), así una voz que pueda que dar un poco seca a la hora de oírla, ganará bastante si se le aplica el efecto *reverb*. Por lo general, para composición musical un ligero "toque" de *reverb* a cada instrumento no viene nada mal.

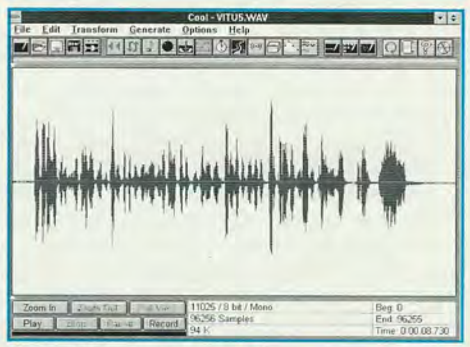
La mayoría de las tarjetas incorporan un procesador de efectos (que suelen ser 2 efectos simultáneos) con *reverb* y *chorus*.

NOISE REDUCTION

Cuando grabamos un sonido a nuestro ordenador, puede que éste tenga algo de ruido al oírlo. Para ver si en una canción grabada o en un efecto de sonido grabado en nuestro ordenador es necesario aplicar este efecto, tenemos que escuchar una parte de la onda en la que no haya sonido, si en esa parte (sin sonido) escuchamos ruido de fondo, entonces es conveniente aplicar este filtro de reducción de ruido.

¡¡CUIDADO!!

No es conveniente abusar de los efectos descritos anteriormente. Si oímos una canción y nos fijamos en el uso de estos efectos, observaremos que el bajo tiene aplicado los efectos de compresor y *reverb*, la guitarra tiene *chorus* y *delay* y que los teclados tienen *reverb*, pero siempre en su justa cantidad. Si pretendemos crear sonidos que luego vamos a utilizar en nuestras composiciones (tanto sonidos de instrumentos como de efectos ambientales) tenemos que tener cuidado de no recargarlos con efectos, ya que el resultado de la mezcla final de dichos sonidos puede saturar demasiado al oyente (por muy bien que estos sonidos suenen por separado).



Onda con efecto de flanger.